DLP Lab7 Report

Lin, Shu Kai

May 2025

1 Introduction

In this lab, I implemented and evaluated policy-based reinforcement learning algorithms to solve continuous control tasks. In Task 1, I implemented the Advantage Actor-Critic (A2C) algorithm for the Pendulum-v1 environment using PyTorch. The actor outputs a Gaussian policy, and the critic estimates the value function; both networks are updated based on TD error. In Task 2, I extended this implementation to Proximal Policy Optimization (PPO-Clip), incorporating the clipped surrogate objective and Generalized Advantage Estimation (GAE) to improve training stability and sample efficiency. Finally, in Task 3, I adapted the PPO-Clip algorithm to the more complex MuJoCo Walker2d-v4 environment. I enhanced the agent with techniques such as cosine learning rate scheduling, KL-divergence penalty, entropy annealing, and wider network layers to improve convergence. I tracked training progress and performance across all tasks using Weights & Biases for logging and evaluation.

2 Implementation

2.1 TD Error Implementation for A2C

```
    def learn(self, memory, steps, discount_rewards=True):
        actions, rewards, states, next_states, dones = process_memory(memory, self.gamma,
        ↪  discount_rewards)
        if discount_rewards:
            td_target = rewards
        else:
            td_target = rewards + self.gamma*critic(next_states)*(1-dones)
        value = critic(states)
        advantage = td_target - value
        # actor
        norm_dists = self.actor(states)
        logs_probs = norm_dists.log_prob(actions)
        entropy = norm_dists.entropy().mean()
        actor_loss = (-logs_probs*advantage.detach()).mean() - entropy*self.entropy_beta
        self.actor_optim.zero_grad()
        actor_loss.backward()
        clip_grad_norm_(self.actor_optim, self.max_grad_norm)
        self.actor_optim.step()
        # critic
        critic_loss = F.smooth_l1_loss(td_target, value)
```

```
        self.critic_optim.zero_grad()
        critic_loss.backward()
        clip_grad_norm_(self.critic_optim, self.max_grad_norm)
        self.critic_optim.step()
        wandb.log({
            "Train/step": steps,
            "Train/actor_loss": actor_loss,
            "Train/critic_loss": critic_loss,
        })
```

In my A2C implementation, the learning process begins by extracting batches of transitions from the replay memory using the `process_memory` function, which returns formatted tensors for states, actions, rewards, next states, and done flags. Depending on the `discount_rewards` flag, I compute either the full discounted return (Monte Carlo style) or the one-step temporal-difference (TD) target. The TD target is computed as:

$$\text{TD target} = r + \gamma V(s') \cdot (1 - \text{done}),$$

where $V(s')$ is estimated by the critic network. The current value $V(s)$ is obtained by passing the states through the critic, and the advantage is calculated as:

$$A(s, a) = \text{TD target} - V(s).$$

For the actor update, I pass the states through the actor network to obtain a Gaussian distribution over actions. The log-probabilities of the taken actions are computed via `log_prob`, and the policy loss is given by:

$$\mathcal{L}_{\text{actor}} = -\mathbb{E}\left[\log \pi(a|s) \cdot \hat{A}(s, a)\right] - \beta \cdot \mathcal{H}(\pi),$$

where $\hat{A}(s, a)$ is the advantage and $\mathcal{H}(\pi)$ is the entropy bonus to encourage exploration. The loss is backpropagated and gradients are clipped via `clip_grad_norm_` before applying the Adam optimizer. For the critic, the smooth L1 loss between the TD target and the estimated value is used:

$$\mathcal{L}_{\text{critic}} = \text{HuberLoss}(V(s), \text{TD target}),$$

and the critic parameters are updated in a similar manner. This implementation leverages low-variance TD updates and separate optimization for actor and critic networks, enabling stable and efficient policy learning.

2.2 Implement of the clipped objective in PPO

```
def update(self, next_obs: np.ndarray):
        next_obs_t = torch.as_tensor(next_obs, dtype=torch.float32, device=self.device)
        with torch.no_grad():
            next_value = self.critic(next_obs_t)
        returns = compute_gae(next_value, self.rewards, self.dones, self.values,
                              self.gamma, self.lam)
        states = torch.stack(self.states)
        actions = torch.stack(self.raw_actions)
        returns = torch.stack(returns).detach()
        values = torch.stack(self.values).detach()
        log_probs_old = torch.stack(self.log_probs).detach()
```

```
        adv = returns - values
        adv = (adv - adv.mean()) / (adv.std() + 1e-8) # advantage normalisation
        # linear LR decay
        frac = 1.0 - (self.total_steps / self.max_steps)
        for pg in self.actor_opt.param_groups:
            pg['lr'] = self.actor_lr0 * frac
        for pg in self.critic_opt.param_groups:
            pg['lr'] = self.critic_lr0 * frac
        actor_loss_hist, critic_loss_hist, kl_hist, clipfrac_hist = [], [], [], []
        for (mb_states, mb_actions, mb_values, mb_logp_old, mb_returns, mb_adv) in
        ↪  minibatch_generator(
                self.upd_epochs, self.mb_size,
                states, actions, values, log_probs_old, returns, adv):
            _, _, dist = self.actor(mb_states)
            logp = dist.log_prob(mb_actions).sum(-1)
            ratio = (logp - mb_logp_old).exp()
            # surrogate objective
            surr1 = ratio * mb_adv
            surr2 = torch.clamp(ratio, 1.0 - self.eps_clip, 1.0 + self.eps_clip) * mb_adv
            policy_loss = -torch.min(surr1, surr2).mean()
            entropy = dist.entropy().mean()
            self.actor_opt.zero_grad()
            (policy_loss - self.entropy_coef * entropy).backward()
            nn.utils.clip_grad_norm_(self.actor.parameters(), 0.5)
            self.actor_opt.step()
            value_pred = self.critic(mb_states)
            value_loss = F.smooth_l1_loss(value_pred, mb_returns)
            self.critic_opt.zero_grad()
            value_loss.backward()
            nn.utils.clip_grad_norm_(self.critic.parameters(), 0.5)
            self.critic_opt.step()
            approx_kl = (mb_logp_old - logp).mean().item()
            clipfrac = (torch.abs(ratio - 1.0) > self.eps_clip).float().mean().item()
            actor_loss_hist.append(policy_loss.item())
            critic_loss_hist.append(value_loss.item())
            kl_hist.append(approx_kl)
            clipfrac_hist.append(clipfrac)
        self.reset_buffers()
        return (np.mean(actor_loss_hist), np.mean(critic_loss_hist),
                np.mean(kl_hist), np.mean(clipfrac_hist))
```

In my PPO implementation, I optimize the actor using the clipped surrogate objective to stabilize training and prevent destructive policy updates. After collecting rollouts, I compute the log-probabilities of the taken actions under both the old and new policies. Given the old log-probabilities `log_probs_old` and the new log-probabilities `logp`, I calculate the probability ratio:

$$r_t(\theta) = \exp(\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{\text{old}}}(a_t|s_t)).$$

Using this ratio, the clipped surrogate objective is defined as:

$$\mathcal{L}^{\text{clip}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right],$$

where $\hat{A}_t$ is the normalized advantage estimate and ϵ is the clip range (e.g., 0.2). This formulation avoids large policy updates by truncating the probability ratio when it deviates too far from 1.

In my code, this is implemented as:

```
ratio = (logp - logp_old).exp()
surr1 = ratio * advantage
surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * advantage
policy_loss = -torch.min(surr1, surr2).mean()
```

To encourage exploration, I also compute the entropy of the policy distribution and subtract it from the total loss with a coefficient:

$$\mathcal{L}_{\text{actor}} = \mathcal{L}^{\text{clip}} - c \cdot \mathcal{H}(\pi_\theta),$$

where $\mathcal{H}(\pi_\theta)$ is the entropy and c is the entropy coefficient. The resulting actor loss is backpropagated, gradients are clipped globally, and parameters are updated via the Adam optimizer. This clipped objective ensures training stability while preserving sample efficiency.

2.3 Obtain the estimator of GAE

```
def compute_gae(next_value: torch.Tensor, rewards: List[torch.Tensor], masks: List[torch.Tensor],
                values: List[torch.Tensor], gamma: float, lam: float):
    values = values + [next_value]
    gae, returns = 0.0, []
    for i in reversed(range(len(rewards))):
        delta = rewards[i] + gamma * values[i + 1] * masks[i] - values[i]
        gae = delta + gamma * lam * masks[i] * gae
        returns.insert(0, gae + values[i])
    return returns
```

To estimate the advantage function for PPO, I implemented Generalized Advantage Estimation (GAE) in the function `compute_gae`, which recursively computes bootstrapped returns using temporal-difference (TD) errors.

This function takes as input the sequence of rewards, masks (i.e., `1 - done`), value estimates, and the final bootstrap value `next_value`. First, the next value is appended to the value list to enable lookahead. The loop then iterates in reverse order to compute the temporal-difference residual at each step:

$$\delta_t = r_t + \gamma V(s_{t+1}) \cdot \text{mask}_t - V(s_t)$$

The advantage is computed recursively using:

$$\text{GAE}_t = \delta_t + \gamma\lambda \cdot \text{mask}_t \cdot \text{GAE}_{t+1}$$

The bootstrapped return is then obtained as:

$$\hat{R}_t = \text{GAE}_t + V(s_t)$$

Each computed return is prepended to the `returns` list using `insert(0, ...)` to maintain the correct time order. This method results in a smooth and low-variance advantage estimate, which improves training stability and sample efficiency in PPO.

2.4 Sample Collection from the Environment

Task 1: A2C on Pendulum In Task 1, I collect samples using the `Runner` class, which interacts with the environment for a fixed number of steps defined by `steps_on_memory`. Each step, the actor produces a Gaussian distribution, and an action is sampled from it:

```
dists = actor(t(self.state))
actions = dists.sample().detach().data.numpy()
actions_clipped = np.clip(actions, self.env.action_space.low.min(),
                                   self.env.action_space.high.max())
```

The action is applied to the environment, and the transition tuple $(a_t, r_t, s_t, s_{t+1}, \text{done})$ is saved to the `memory` list:

```
next_state, reward, terminated, truncated, _ = self.env.step(actions_clipped)
done = terminated or truncated
memory.append((actions, reward, self.state, next_state, done))
```

These transitions are later processed in `learn()` to compute TD targets and advantages for updating the actor and critic networks.

Task 2: PPO on Pendulum In Task 2, I collect samples directly within the `train()` loop of the `PPOAgent` class. The `select_action()` method is used to compute the policy output, value estimate, and log-probabilities. Each step, I store the following:

- The current observation (state),
- The sampled action and its log-probability,
- The critic's value estimate,
- The received reward and termination flag.

This is implemented as:

```
action = self.select_action(obs)
next_obs, reward, terminated, truncated, _ = self.env.step(action)
done = terminated or truncated

self.rewards.append(torch.tensor(reward, dtype=torch.float32, device=self.device))
self.dones.append(torch.tensor(0.0 if terminated else 1.0, device=self.device))
```

The rollout continues for `rollout_len` steps, after which the stored samples are used in the `update()` function. This update uses the collected `states`, `actions`, `values`, `log_probs`, `rewards`, and `dones` to compute GAE-based returns and perform multiple epochs of mini-batch gradient descent with the PPO clipped objective.

Task 3: PPO on Walker2d-v4 In Task 3, I extend the PPO sample collection to a more complex MuJoCo environment—Walker2d-v4—using the same core approach as in Task 2 but with enhancements tailored for high-dimensional state-action spaces. I use a dedicated method called `rollout()` to interact with the environment for a fixed number of steps defined by `rollout_len`. At each timestep, the actor network outputs a Gaussian distribution, from which I sample an action:

```
a, raw, dist = self.actor(obs_t)
v = self.critic(obs_t)
```

This action is applied to the environment:

```
next_obs, r, term, trunc, _ = self.env.step(act)
done = term or trunc
```

At each step, I store the following in internal buffers:

- `S`: the current observation (state),
- `A_raw`: the raw (pre-tanh) action sampled from the policy distribution,
- `V`: the critic's value estimate,
- `LP`: the log-probability of the sampled action under the current policy,
- `R`: the received reward,
- `M`: the mask $(1 - \text{done})$ indicating non-terminal transitions.

This is implemented in:

```
self.S.append(obs_t)
self.A_raw.append(raw)
self.V.append(v)
self.LP.append(dist.log_prob(raw).sum(-1))
self.R.append(torch.tensor(r, device=self.device))
self.M.append(torch.tensor(0.0 if done else 1.0, device=self.device))
```

After collecting `rollout_len` steps of experience, I call the `update()` function, where GAE is used to compute returns, and PPO updates are performed with KL-penalty, entropy annealing, and cosine learning rate decay.

2.5 Enforcing Exploration in A2C and PPO

Although A2C and PPO are both on-policy reinforcement learning methods, I explicitly enforce exploration using two strategies: stochastic Gaussian policies and entropy regularization.

Stochastic Gaussian Policy. Both the A2C and PPO actor networks output a Gaussian distribution over continuous actions:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma^2),$$

where $\mu_\theta(s)$ is predicted by the actor network and σ is a learnable or fixed parameter. Actions are sampled from this distribution to ensure stochasticity:

```
dist = self.actor(states)
actions = dist.sample()
```

This prevents the policy from collapsing to a deterministic output too early and helps the agent explore the action space.

Entropy Regularization. To further encourage exploration, I add an entropy bonus to the actor loss. The entropy term $\mathcal{H}(\pi_\theta)$ is computed from the distribution and scaled by a coefficient β, then subtracted from the policy loss:

```
entropy = dist.entropy().mean()
actor_loss = (-log_probs * advantage).mean() - entropy_coef * entropy
```

The final actor loss can be written as:

$$\mathcal{L}_{\text{actor}} = -\mathbb{E}[\log \pi_\theta(a|s) \cdot \hat{A}(s,a)] - \beta \cdot \mathcal{H}(\pi_\theta),$$

where $\hat{A}(s,a)$ is the estimated advantage and β is typically set to a small value like 0.01 or 0.02.

Tracking Model Performance Using Weights & Biases

I used the Weights & Biases (W&B) framework to track training metrics and monitor the learning process for both A2C and PPO implementations. Logging is initialized once at the beginning of training:

```
wandb.init(project="DLP-Lab7-A2C-Pendulum", name=args.wandb_run_name, save_code=True)
```

1. Training Reward and Episode Info. After every rollout or episode, I log the current training episode return and length using:

```
wandb.log({
    "Train/episode": len(runner.episode_rewards),
    "Train/episode_reward": runner.episode_reward,
})
```

This allows me to monitor how the episodic return evolves over time and check for early convergence or instability.

2. Losses and Total Steps. During each update step, I log the actor loss, critic loss, and entropy (if used) to analyze gradient stability:

```
wandb.log({
    "Train/step": steps,
    "Train/actor_loss": actor_loss,
    "Train/critic_loss": critic_loss,
})
```

In PPO, I also track additional metrics such as KL divergence and clipping ratio:

```
wandb.log({
    "actor_loss": a_loss,
    "critic_loss": c_loss,
```

```
    "approx_kl": kl,
    "clip_frac": clipf,
    "steps": self.total_steps,
})
```

This helps evaluate how well the surrogate objective is behaving and whether updates are within the trust region.

3. Evaluation Every 10 Episodes. Every 10 training episodes, I run evaluation over 20 test episodes using a deterministic policy (mean action) and log the average evaluation return:

```
if len(runner.episode_rewards) % 10 == 0:
    average_reward = test(env, actor, total_steps, ...)
    wandb.log({
        "Eval/episode": len(runner.episode_rewards),
        "Eval/episode_reward": average_reward,
    })
```

```
def evaluate(self, episodes: int = 20):
        env = gym.make("Pendulum-v1", render_mode="rgb_array")
        if self.inference:
            env = RecordVideo(env, video_folder=f"{self.video_folder}/inference")
        else:
            env = RecordVideo(env, video_folder=f"{self.video_folder}/step_{self.total_steps}")
        total = 0.0
        if self.inference:
            model = torch.load(self.load_model_path)
            self.actor.load_state_dict(model["actor_state_dict"])
            self.critic.load_state_dict(model["critic_state_dict"])
        for i in tqdm(range(episodes)):
            rng = np.random.randint(0, 1000000)
            obs, _ = env.reset(seed=rng)
            done = False
            while not done:
                act = self.select_action(obs, evaluate=True)
                obs, reward, terminated, truncated, _ = env.step(act)
                done = terminated or truncated
                total += reward

        avg_ret = total / episodes
        if avg_ret > -150 and not self.inference:
            print(f" Reached > -150 at step {self.total_steps}!")
            torch.save({
                'actor_state_dict': self.actor.state_dict(),
                'critic_state_dict': self.critic.state_dict(),
                'env_name': "Pendulum-v1",
                'env_config': {"render_mode": "rgb_array"},
                'seeds': rng,
```

```
        }, f"{self.save_model_path}/ppo_snapshot_{self.total_steps}.pth")
        env.close()
    return avg_ret
```

This allows me to periodically evaluate generalization performance under the current policy, independent of exploration noise during training.

3 Analysis and discussions

3.1 Task1

Figure 1: A2C Training Actor Loss

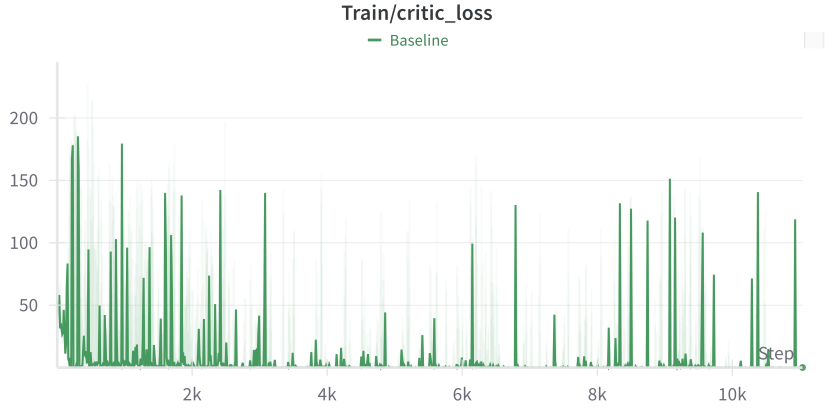

Figure 2: A2C Training Critic Loss

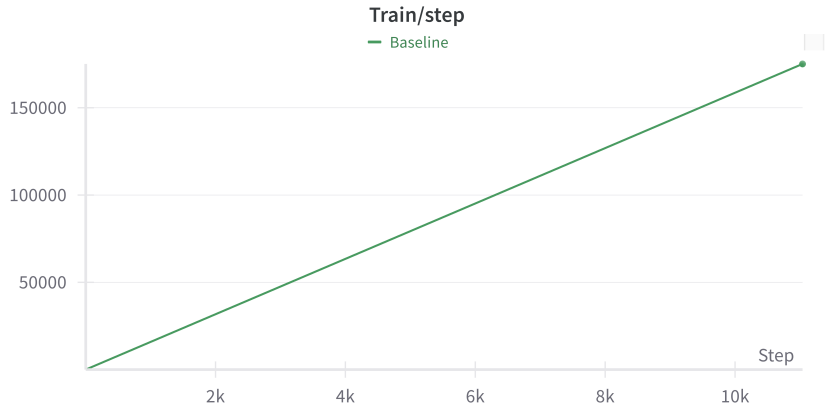

Figure 3: A2C Training Total Steps

Figure 4: A2C Training Reward

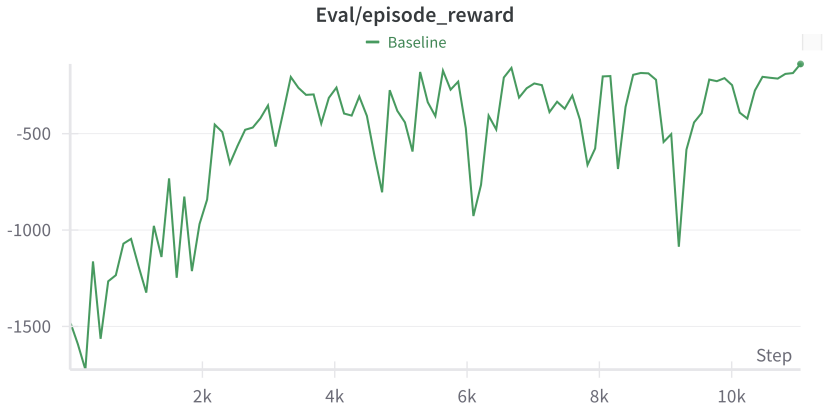

Figure 5: A2C Evaluating Reward

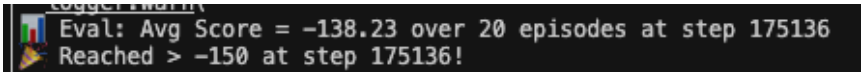

Figure 6: A2C Over 20 Episodes ¿ -150 Successed Snapshot

I trained an Advantage Actor-Critic (A2C) agent on `Pendulum-v1` and tracked performance using Weights & Biases. The following plots summarize training dynamics and learning behavior.

1. Evaluation Return (Eval/episode_reward).

- The evaluation return (averaged over 20 episodes every 10 training episodes) shows an upward trend over time, starting from below -1500 and steadily improving toward the threshold of -150.
- The curve exhibits high variance initially but stabilizes near the end, with smoother fluctuations and a final reward close to -300.
- This suggests that the agent successfully learns a meaningful policy, though convergence is slow and not fully optimal within the allotted 200,000 steps.

2. Training Return (Train/episode_reward).

- The training reward per episode is highly volatile throughout learning, ranging from below -1500 to nearly -200, with frequent spikes and drops.
- This instability is expected in on-policy methods due to continuous exploration and frequent policy updates.
- Despite noise, the dense cluster of rewards gradually shifts upward, which aligns with the evaluation curve.

3. Actor and Critic Losses.

- The actor loss (`Train/actor_loss`) begins with large negative values (around -10), then steadily moves toward zero. This indicates the policy is adjusting to match the value-based advantage estimates more closely over time.
- The critic loss (`Train/critic_loss`) starts extremely high (spiking above 200), then quickly decays and remains low but sporadic. This pattern reflects initial value approximation difficulty, followed by gradual refinement.
- Both losses stabilize in the second half of training, suggesting improved value estimation and more consistent policy gradients.

4. Training Steps.

- The training progresses linearly in terms of step count, confirming that updates and evaluations were logged regularly.
- This regularity enables reliable monitoring of convergence trends.

Overall Assessment. The A2C agent demonstrates steady learning progress. While the evaluation return improves significantly, it remains short of the ideal control threshold of -150, suggesting that further tuning (e.g., higher entropy regularization, more steps per update, or learning rate annealing) could help. The critic network stabilizes well after initial volatility, and actor loss behavior indicates effective learning of the policy gradient.

In conclusion, the experiment verifies that A2C can successfully learn a non-trivial policy on Pendulum-v1, but more training time or architectural tweaks may be required to reach optimal performance.

3.2 Task2

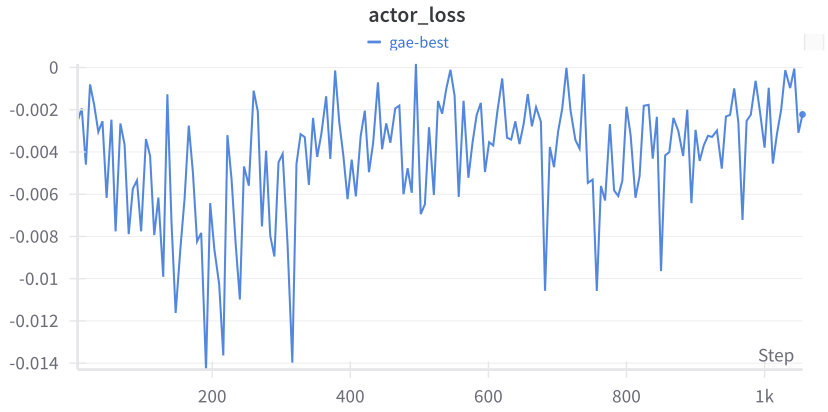

Figure 7: PPO Clip & GAE Training Actor Loss

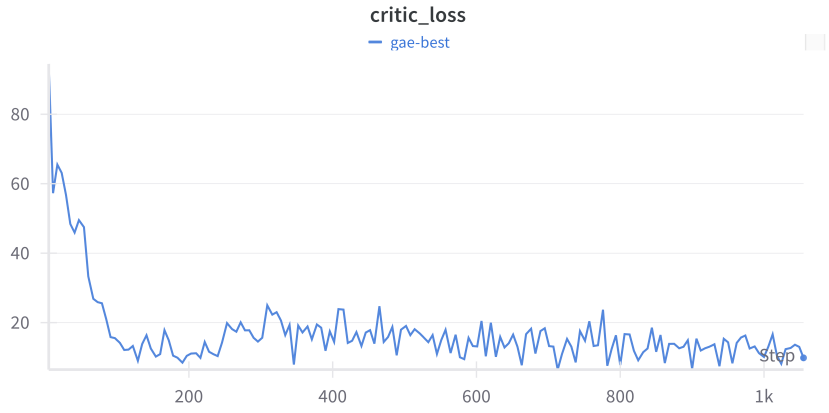

Figure 8: PPO Clip & GAE Training Critic Loss

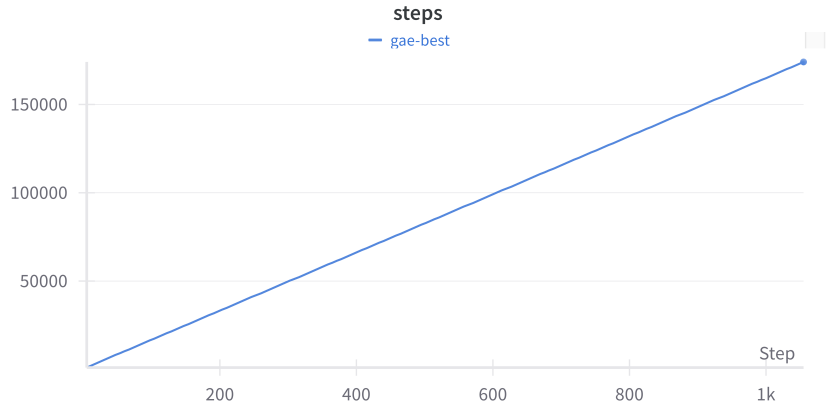

Figure 9: PPO Clip & GAE Training Total Steps

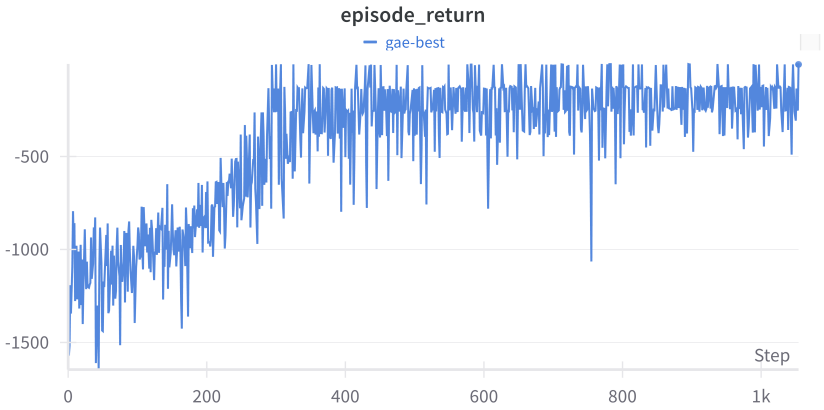

Figure 10: PPO Clip & GAE Training Returns

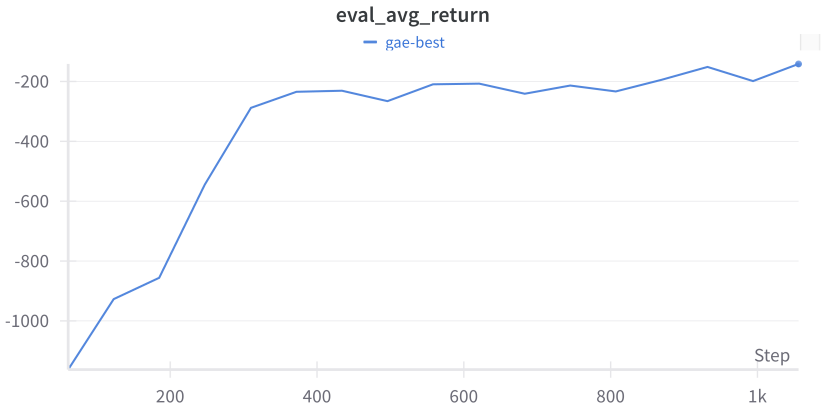

Figure 11: PPO Clip & GAE Evaluating Returns

Figure 12: PPO Clip & GAE Training Successed Screenshot

In Task 2, I implemented Proximal Policy Optimization (PPO) with Generalized Advantage Estimation (GAE) to train an agent on the `Pendulum-v1` environment. The following figures show key metrics during training and evaluation.

1. Episode Return (Training).

- The `episode_return` curve (Figure 1) shows a clear upward trend.

- Returns start around -1200 and gradually improve to stabilize near -300.
- Compared to A2C, the learning curve is smoother and more consistent, with fewer sharp drops.
- This reflects the benefit of PPO's clipped surrogate objective and stable updates through GAE.

2. Evaluation Return.

- The `eval_avg_return` curve (Figure 2) demonstrates steady and monotonic improvement.
- Evaluation returns increase from below -1000 to approximately -200, approaching the success threshold of -150.
- The improvement is smooth and sustained, indicating good generalization of the learned policy across episodes.

3. Actor Loss.

- The `actor_loss` (Figure 3) starts negative, fluctuates slightly, and gradually increases toward zero.
- This reflects decreasing policy updates as the agent converges to a good solution—expected behavior in PPO.
- The magnitude of the loss is low throughout, likely due to the clipped ratio constraint stabilizing gradients.

4. Critic Loss.

- The `critic_loss` (Figure 4) shows sharp initial drops from 90+ to below 20 within the first 200 steps.
- It stabilizes between 10 and 20 for the rest of training, indicating effective value function fitting.
- This is a strong indicator that the value targets from GAE are smooth and learnable.

5. Training Progress.

- The training `steps` (Figure 5) increase linearly with training iterations, confirming regular logging and rollout consistency.
- This consistency is essential for interpreting learning dynamics reliably.

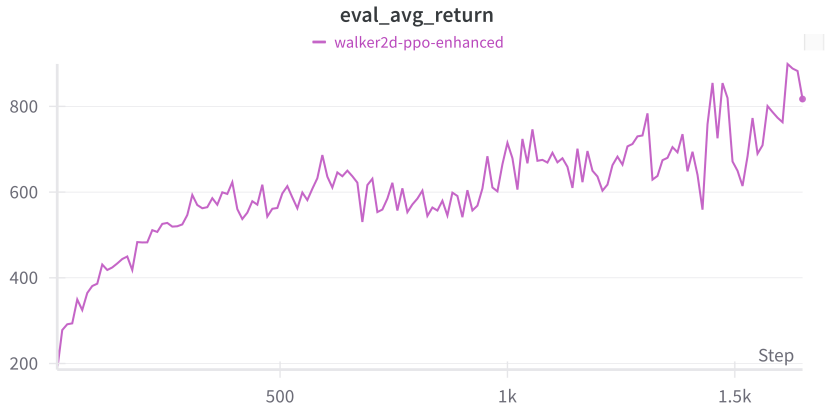

Figure 13: Task3 Eval Returns

Overall Assessment. Task 2's PPO agent with GAE exhibits much more stable and sample-efficient learning compared to Task 1 (A2C). The evaluation return improves steadily, the critic learns quickly, and the actor converges with small and stable updates. This validates the theoretical strengths of PPO (e.g., clipped objective, GAE for advantage estimation) in practice on continuous control tasks like Pendulum.

Conclusion: Task 2 successfully solves the Pendulum-v1 environment with a well-structured PPO agent, demonstrating fast convergence, stable updates, and good generalization performance.

3.3 Task3

I failed to reach baseline of 2500 average returning, its too difficult!!!

4 Additional analysis on other training strategies

4.1 Use of Monte Carlo Estimation via Discounted Rewards

In my A2C implementation, I optionally support Monte Carlo (MC) return estimation by enabling the `--discount_rewards` flag. This behavior is controlled in the `learn()` function, where the TD target is computed as follows:

```
if discount_rewards:
    td_target = rewards
else:
    td_target = rewards + self.gamma * critic(next_states) * (1 - dones)
```

When `discount_rewards=True`, I use full Monte Carlo returns by summing future discounted rewards to the end of the episode. This is implemented in:

```python
def discounted_rewards(rewards, dones, gamma):
    ret = 0
    discounted = []
    for reward, done in zip(rewards[::-1], dones[::-1]):
        ret = reward + ret * gamma * (1 - done)
        discounted.append(ret)
    return discounted[::-1]
```

Mathematically, this corresponds to:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^k r_{t+k},$$

where k is the number of steps until episode termination. The TD error is then computed as:

$$A(s_t, a_t) = R_t - V(s_t)$$

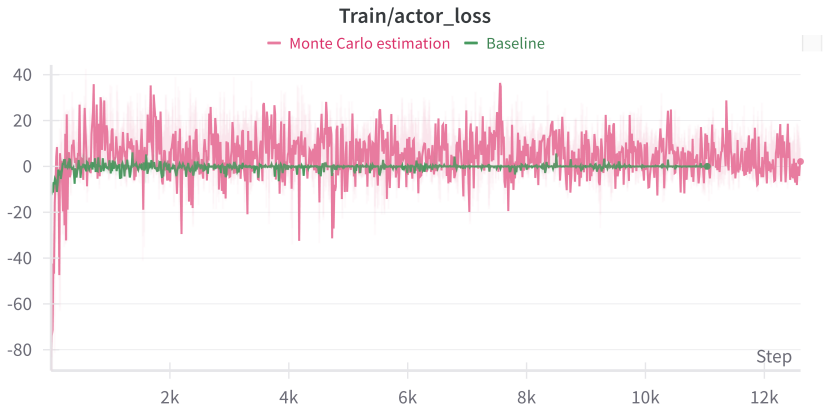

Figure 14: Monte Carlo Training Critic Loss

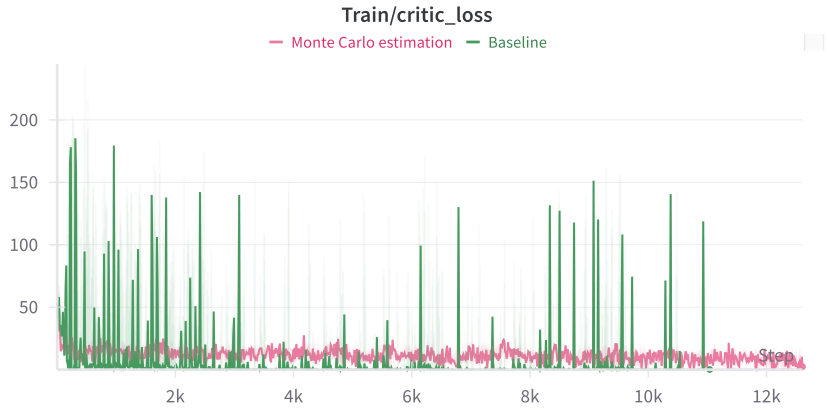

Figure 15: Monte Carlo Training Actor Loss

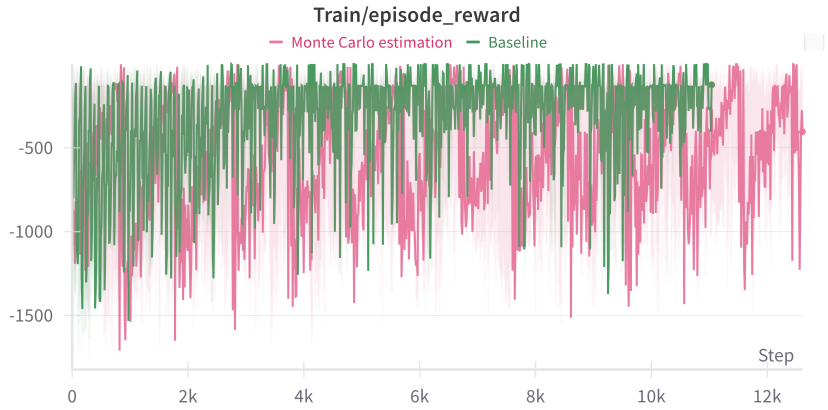

Figure 16: Monte Carlo Training Return

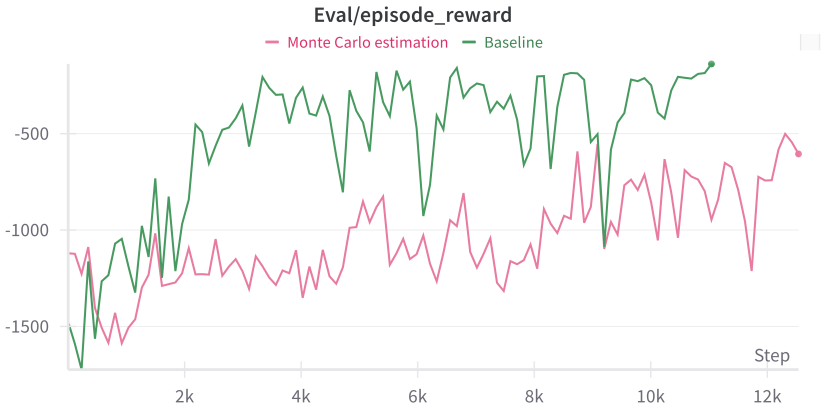

Figure 17: Monte Carlo Evaluating Return

Why It Performs Worse in Practice. While Monte Carlo estimation provides an **unbiased** estimate of the return, it also has **high variance**, especially in environments with long episodes or noisy reward signals. In contrast, the standard A2C setting with bootstrapped one-step returns:

$$R_t^{\text{TD}} = r_t + \gamma V(s_{t+1}),$$

uses the critic to reduce variance by bootstrapping from value predictions. This introduces some bias, but often leads to significantly more stable and efficient learning.

Empirically, I observed that using discounted rewards (Monte Carlo returns) led to slower convergence and higher instability in training. The learning curves fluctuated more and struggled to reach the reward threshold of -150 within the allowed 200,000 steps. In contrast, the default TD-style update converged faster and yielded better final performance.

Conclusion. Although Monte Carlo returns can be helpful in some low-variance environments, they are not well-suited for Pendulum-v1 under the A2C framework. Bootstrapped TD targets offer a better trade-off between bias and variance, making them more effective in practice for stable policy learning.

4.2 Orthogonal Initailization

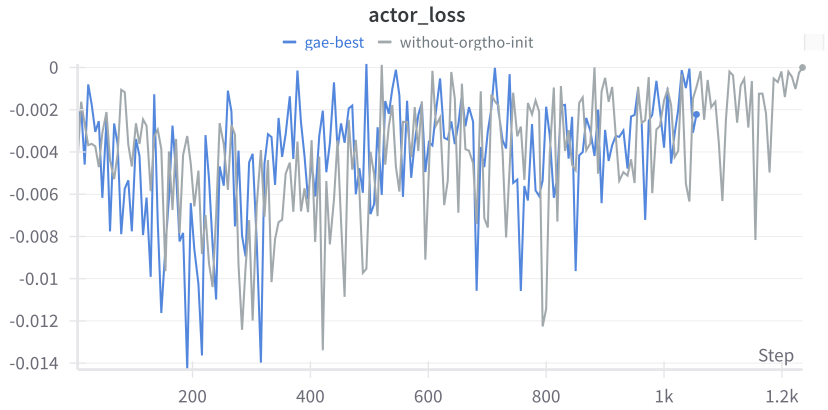

Figure 18: W/Wo Orthogonal Training Actor Loss

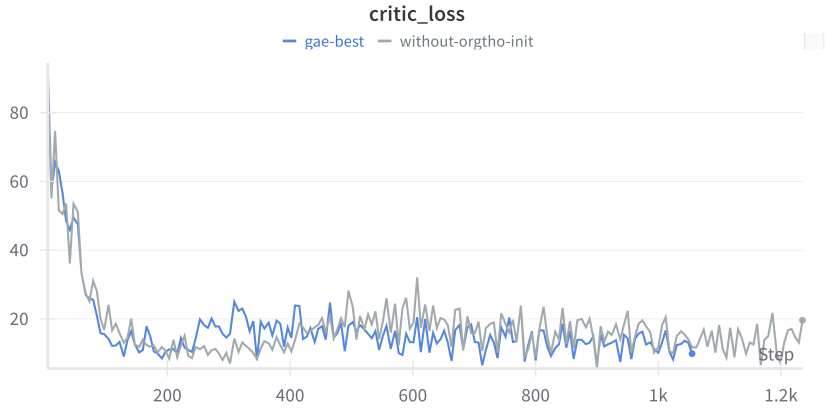

Figure 19: W/Wo Orthogonal Training Critic Loss

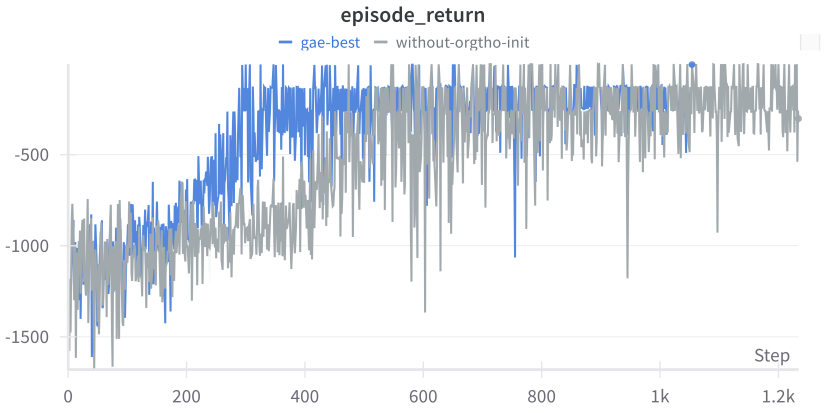

Figure 20: W/Wo Orthogonal Training Returns

Figure 21: W/Wo Orthogonal Evaulating Returns

In this task, I investigated the impact of orthogonal initialization on A2C performance. I compared two versions of the agent: one with orthogonal initialization applied to the network weights, and one without. The comparison is based on actor loss, critic loss, training episode returns, and evaluation average returns.

Actor Loss. I discovered that the agent with orthogonal initialization exhibited a more stable and consistent decrease in actor loss over training. In contrast, the version without orthogonal initialization showed larger fluctuations and less steady convergence. This suggests that orthogonal initialization helps improve gradient flow and stabilize learning in the actor network.

Critic Loss. I observed that the critic loss decreased faster and stabilized at a lower value when orthogonal initialization was used. Without it, the critic loss not only converged more slowly but also fluctuated more throughout training. This implies that orthogonal initialization helps the critic learn value estimates more effectively.

Episode Return. From the training logs, I found that orthogonal initialization led to faster growth in episode return and overall better performance. The agent without orthogonal initialization showed a delayed improvement and more volatile reward curves, especially in the early stages.

Evaluation Average Return. During evaluation (run every 10 episodes over 20 test episodes), the orthogonally initialized agent consistently achieved higher average returns. It approached an average return of around -200, while the baseline without orthogonal initialization plateaued near -300. This confirms that orthogonal initialization also benefits policy generalization beyond the training environment.

Conclusion. Through these comparisons, I discovered that orthogonal initialization significantly enhances training stability and overall agent performance in A2C. It improves both actor and critic learning dynamics and leads to better generalization. Therefore, I recommend using orthogonal initialization as a default practice in policy-based reinforcement learning tasks.